Application of Safety Verification Methodology Framework During Software Development Phases

Prof. K. Amarendra[1]
[1] Vice Principal, Department of CSE,
Dadi Institute of Engineering & Technology,
NH – 5, Anakapalle, Visakhapatnam – 531002, INDIA
viceprincipal@diet.edu.in

Abstract – In order to detect and prevent faults, researchers have developed safety standards, safety analysis techniques, and fault-tolerant techniques; however, there are still no methodology frameworks for verifying safety-critical software systems. This research's methodology combines software-safety methods into a comprehensive whole for the purpose of verifying safety-critical software systems. This research concentrated on developing a methodology framework that combines static-verification, dynamic-verification, and fault-tolerant concepts for verifying safety-critical software systems.

Keywords: *Safety critical systems, Verification & Validation, SVVM, Software Development Life Cycle.*

*****

1. INTRODUCTION

Ensuring the correctness of computer systems is a complex task of paramount importance, especially when such systems control and monitor life-critical operations. The verification of industrial computer systems is particularly difficult due to their size and complexity. The most frequently used methods, simulation and testing, are not exhaustive and can miss important errors. While the use of both methods can increase the reliability of the application, they cannot fulfill the verification needs of modern complex safety-critical systems. Formal methods are an additional methodology to tackle this problem. Formal verification tools allow an exhaustive search to be automatically performed on the state space of the system, avoiding the shortcomings of both simulation and testing.

The increase in software-controlled systems is due to many factors such as cost, flexibility, and reliability. Research in software safety falls into two categories: (1) improving software safety before releasing the product by using verification techniques and (2) improving software safety after releasing the product by using fault-tolerant techniques. For verification techniques, most researchers concentrate on static methods, which analyze a software system's safety without executing it, and ignore dynamic methods, which analyze a system's safety by executing it. This research concentrated on developing a methodology framework that combines static-verification, dynamic-verification, and fault-tolerant concepts for verifying safety-critical software systems.

This research's methodology combines software-safety methods into a comprehensive whole for the purpose of verifying safety-critical software systems. For clarification purposes, this document treats the words approach, technique, and method as having synonymous definitions. A methodology brings structure, guidance, and specific techniques all together in order to improve a given process - in this case, the process is software-safety verification. This research deals with Safety Verification and validation Methodology (SVVM). Below diagram represents Standard phases for System development showing general exit and entry conditions.

Deficiencies within software safety: As normal for relatively new fields, software-safety methods and practices have deficiencies in many areas. Within software engineering, researchers have been looking into safety-related issues for approximately the past decade. Their research focused mainly on techniques for statically verifying and modeling safety-critical software systems and providing standards for developing such systems. However, safety standards are often too vague and have few and scattered guidelines.

This Safety Verification and Validation Methodology (SVVM) Framework includes the following Activities:

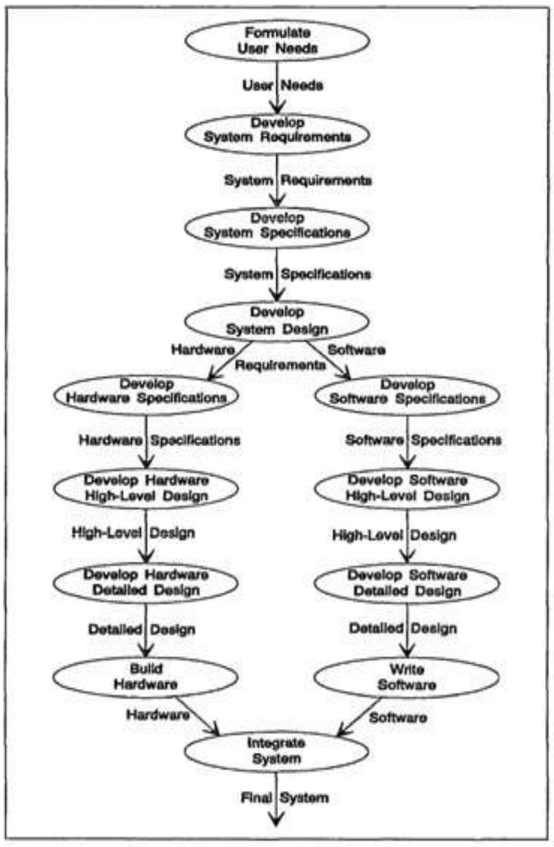
1) Identify appropriate life-cycle phases

2) Define entry and exit criteria for each phase

3) Define activities to occur during each phase (inner-phase activities)

4) Specify techniques for doing each inner-phase activity

5) Specify documentation approaches for each inner-phase activity



2. PRELIMINARY SOFTWARE HAZARD ANALYSIS

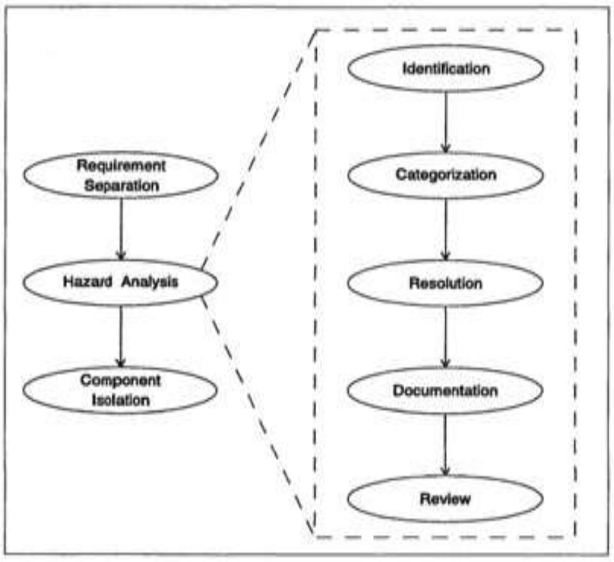
Requirement Separation: The first activity that occurs during the system-design phase is separating hardware- and software-related requirements by using the system specification and design since these documents specify which components are software controlled. Requirement separation is necessary in order to identify software components, divide up the work load, and enable a systematic means for safety analysis - analogous activities take place for hardware.

*Software Hazard Analysis:*As Figure shows, preliminary software hazard analysis involves several activities. Before introducing these activities, this paragraph examines the safety problem (i.e., what developmental areas cause the most hazards). The main causes for safety problems are specification errors. In fact, many errors trace back to specification documents according to current research. Furthermore, Jaffe and Leveson point out that most software failures are due to specification incompleteness.Therefore, in order to improve system safety, preliminary software hazard analysis attempts to insure a complete, precise, and correct software-safety specification.

Identification: To reduce specification errors, hazard identification becomes a crucial first step when developing and analyzing safety-critical systems. Naturally, there are other reasons for hazards occurring, but the fact remains that engineers cannot protect a system against unidentified hazards with any realistic confidence level. Some feel that hazard identification is a "relatively easy" process;** however, this categorization over simplifies the hazard-identification process, which is an area that still needs further research.

In order to identify software hazards, engineers must consider all informational sources relating to the individual software components such as system specifications, historical data, system-level hazard lists, hardware-interface specifications, human experts, and system prototypes and models.

Categorization: After identifying a software hazard, the developer must determine the hazard's categorization (i.e., severity). A hazard's categorization represents the worst-case consequence that it can cause. A software hazard's categorization along with other factors determines the rigor necessary during development to insure that the event will not occur.



Resolution: For each software hazard, the analysis team determines if they can or should eliminate the hazard altogether. Hazard elimination can occur in two ways: (1) move the hazard or (2) remove the hazard's consequence. Moving a software hazard involves making the hardware (or other system component) solely responsible for the respective system requirement and requires a specification change. Reasons for moving a hazard can be due to cost issues, safety issues, or both. Removing a hazard's consequence involves eliminating system functionality and requires at least a specification change. If hazard elimination is not appropriate, then the developer must ensure that the likelihood of the hazard occurring is sufficiently low.

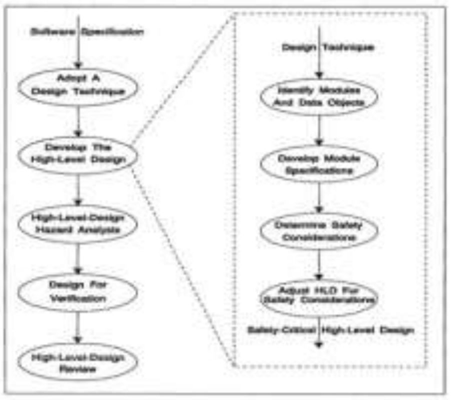
*Documentation:*For every hazard , the developer must document all information relating to the hazard such as its severity, consequences, and description. This documentation provides necessary information for adjusting the system design, preparing the software requirements, and verifying the system. High-severity software hazards, without considering other safety measures, require rigorous development and verification in order to insure that they do not occur.

Review: After completing the preliminary software hazard analysis, the developer must review the process and resulting documentation in order to insure that the process was complete, correct, and precise. The review process by nature depends almost entirely on human skill; therefore, human error and oversight is always a possibility.

*Component Isolation:*An issue related closely to partitioning critical and non-critical components is physical isolation, which should be a part of the hardware and software requirements (i.e., the output from the system-design phase). Physical isolation deals with separating critical and non-critical components not in documentation terms, as partitioning does, but in design terms. For example, keeping critical software on a board by itself so that it has its own processor, memory, etc.

3. HIGH LEVEL DESIGN HAZARD ANALYSIS

Developing the High-Level Design: After choosing a design technique, the design team develops the high-level design. Figure shows a process diagram for the high-level-design activities, which may vary slightly depending on the actual design technique that the team chose. The first activity involves identifying all modules and data objects: A module, for example, can be a subroutine, function, task, or ada package. The second activity concentrates on developing module specifications while the third activity analyzes the safety considerations for each module and data object. Finally, the last activity modifies the design in order to adjust it for the safety considerations.

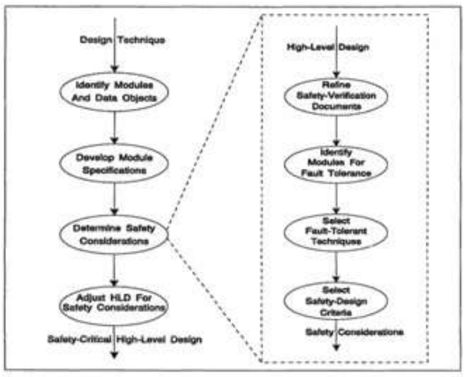

Software Specification
Adopt A Design Technique
Develop The High-Level Design
High-Level-Design Hazard Analysis
Design For Verification
High-Level-Design Review
Design Technique
Identify Modules And Data Objects
Develop Module Specifications
Determine Safety Considerations
Adjust HLD For Safety Considerations
Safety-Critical High-Level Design

Module Identification: A main activity when developing a high-level design is to identify all modules and data objects. Each design technique has its own method for module identification and classification, so this chapter does not cover these details. Data objects can be external inputs or outputs as well as internal data items such as flags, structures, parameters, etc. For large systems, module and object identification can become a difficult task, so the design team should break up large systems into individual components or sub-systems. Then, the design team can proceed with the module identification process. Remember that design is usually an iterative process (i.e., most design teams will not identify all modules correctly on the first pass).

Module Specification: During or after module identification, the design team must specify each module and data object. For modules, these specifications describe their functionality (purpose), input and output parameters, returned values for functions, and all preconditions and postconditions. For data objects, the specification describes their meaning, type, range, constraints, and other relevant information that the specific design technique might require. Module parameters are data objects that require extra information such as their position in the parameter list; whether they are input, output, or input and output parameters; and whether or not they are optional parameters. Once again, due to iterative development, these specifications may require multiple passes before they are complete, correct, and precise.

If the software specification document does not stipulate the method for module and object specification, then the design team must adopt a specification method. There are two broad methods for specifying modules and their objects: (1) informal and (2) formal. An informal module-specification method might use natural-language text, diagrams, pseudo code, or a combination of these items. Formal methods, however, use mathematical notations such as algebraic or lambda-calculus equations: two such methods, which are common in Europe, are VDM and z A Furthermore, formal methods should contain informal descriptions too in order to help support their meaning. Natural-language descriptions are also useful for those individuals who are not familiar with formal methods. Whatever technique the design team chooses, it should allow for correct and precise module specifications so that no ambiguities arise.

Safety Considerations: After identifying all the modules and developing their specifications, the design team should analyze the safety considerations for each module. Figure outlines the safety-consideration process, which involves (1) refining the safety-verification documents, (2) identifying modules that should have fault tolerance, (3) determining and assigning specific fault-tolerant techniques to these modules, and (4) determining and assigning other safety design issues to these modules. For this section, a module may be an Ada package, task, subroutine, or function. In addition to modules, data objects require safety analysis as well in order to insure system safety.


Design Technique
Identify Modules And Data Objects
Develop Module Specifications
Determine Safety Considerations
Adjust HLD For Safety Considerations
Safety-Critical High-Level Design
High-Level Design
Refine Safety-Verification Documents
Identify Modules For Fault Tolerance
Select Fault-Tolerant Techniques
Select Safety-Design Criteria
Safety Considerations

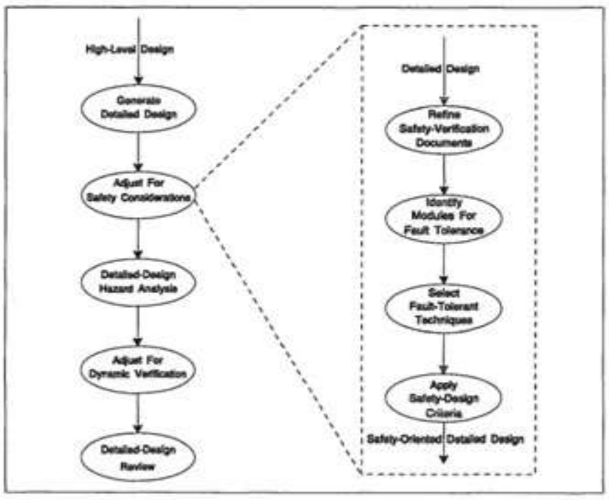
Refine safety documents: Before the design team can recommend specific modules for fault-tolerant designs and before they can effectively assign dynamic-verification requirements, the safety documentation must be up to date. Updating the safety documentation involves identifying modules that can cause software hazards, describing the causes, identifying the relationships between causes, identifying the phases for which a cause is relevant, describing the consequences for each cause, and determining the severity for each cause. In order to determine the modules that can cause software hazards, safety engineers can identify critical data objects in the high-level design and then those modules that create, reference, or transform the data objects.

Selecting module fault tolerance: After identifying the safety-critical modules in the high-level design, the design team must determine and assign appropriate fault-tolerant techniques. If the specification document does not outline the procedure for doing this activity, then the design team must determine the procedure. Currently, there is little research relating fault-tolerant techniques to problem classes. Furthermore, there is no research that says one fault-tolerant technique is better than another.

Adjust High-Level Design: After determining the different safety considerations, the design team must update any affected parts o f the high-level design. The various safety considerations might require changes in parameters, module specifications, and pre- and post-conditions. These considerations might also result in more modules and adjustments to the safety-verification documents.

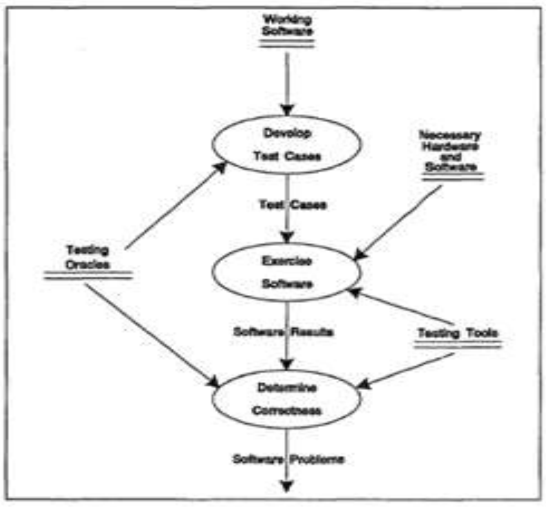
4. DETAILED DESIGN

Detailed-design and code-level hazard analysis involves five activities: (1) generating the detailed design and code, (2) adjusting the design and code for safety considerations, (3) design and code hazard analysis, (4) adjusting the design and code for dynamic verification, and (5) a design and code review. Developing the detailed design is just a further refinement of the high-level design and involves applying the selected design technique at the next level of detail. For object-oriented designs, this refinement means expanding objects to include more detail (i.e., more objects and transformations). For functional-oriented designs, this refinement means expanding actions to include more detail (sub-functions and their data). The refinement process continues until the detailed design is at a low enough level to facilitate coding. While developing the detailed design, the designer may need to create additional modules that are not present in the high-level design.


High-Level Design
Generate Detailed Design
Adjust For Safety Considerations
Detailed-Design Hazard Analysis
Adjust For Dynamic Verification
Detailed-Design Review
Detailed Design
Refine Safety-Verification Documents
Identify Modules For Fault Tolerance
Select Fault-Tolerant Techniques
Apply Safety-Design Criteria
Safety-Oriented Detailed Design

5. DYNAMIC SAFETY VERIFICATION

After static safety verification and general reliability testing, software engineers must dynamically verify the software's safety (i.e., safety testing). As previous chapters mentioned, dynamic safety verification tests the software's safety features by executing the software.



Even if the developer uses rigorous formal techniques to show the software's safety, safety faults may still exist in the system since these techniques do not 1 9 produce 100% reliable software. ' For these reasons, safety testing is important and attempts to show that the software conforms to the safety documentation (e.g., fault trees, event trees, failure modes and effects, etc.), which represent the input-output oracles for the safety-testing process.

6. CASE STUDY DESIGN PROCESS

Railroad Crossing Control System (RCCS)
Crossing gates on a full-size railroads are controlled by a complex control system that causes the gates to be lowered to prevent access to the crossing shortly before a train arrives and to be raised to allow access to resume after the train has departed. This requires the detection of approaching trains or the manual actuation of the crossing gates by an operator. RCCS is a prototype, real-time, safety-critical railroad crossing control system of limited complexity. It is composed of several software-controlled hardware components.

RCCS System Functions:

• Control the overall operation of train on the track circuit.
• Control the opening and closing of Gate 1 and 2 at the railroad intersections
• Control the track lever to change the track route from the outer to the inner loop
• Check the internal health of all the subsystems
• Control the train operation at the Signal Lights
• Monitor all the sensors on the track circuit

RCCS System Operations:
When RCCS is first switched on, the controller does a preliminary check of the normal working status of all the subsystems involved – the driver circuitry, the sensors, the gate assemblies, and the train signals. If all the components are found to be in normal working condition, it executes the code related to normal operation. Initially, the train starts from the platform location and is programmed to run on the outer track. After it completes this cycle, it changes direction and runs on the inner track. This change is facilitated by the track-change level which is present at the intersection of the outer track and inner track. Along the track, the two gates Gate 1 and Gate 2 are automatically lowered when the train nears the railroad intersection and raised when the train leaves the intersection. Whenever the signal lights display Red, the train comes to a halt and resumes running only after a Green signal is given. Whenever the train detects any physical obstacle on the track, the train comes to a halt.

train passes Sensor2 positioned prior to gate, a signal is sent to the controller indicating the approaching train. The controller then sends a signal to the gates assembly, causing the gate arms on either side of the road to close. When the train finally has passed Sensor3, which is positioned just beyond the gate crossing section, a corresponding signal is sent to the controller, which in turn triggers both the gate arms to open simultaneously.

7. SAFETY ANALYSIS AND RESULTS

The safety analysis of RCCS software functions takes place in three sequential steps.

• *Software Failure Mode and Effects Analysis (SFMEA):*
This analysis is performed in order to determine the top events for lower level analysis. SFMEA analysis will be performed following the list of failure types. SFMEA will be used to identify critical functions based on the applicable software specification. The severity consequences of a failure, as well as the observability requirements and the effects of the failure will be used to define the criticality level of the function and thus whether this function will be considered in further deeper criticality analysis. The formulation of recommendations of fault related techniques that may help reduce failure criticality is included as part of this analysis step.

• *Software Fault Tree Analysis (SFTA)*
After determining the top-level failure events, a complete Software Fault Tree Analysis shall be performed to analyse the faults that can cause those failures. This is a top down technique that determines the origin of the critical failure. The top-down technique is applied following the information provided at the design level, descending to the code modules . SFTA will be used to confirm the criticality of the functions

(as output from SFMEA) when analyzing the design and code (from the software requirements phase, through the design and implementation phases) and to help:

- Reduce the criticality level of the functions due to software design and / or coding fault-related techniques used (or recommended to be used)

- Detail the test-case definition for the set of validation test cases to be executed.

• *Evaluation of Results :*The evaluation of the results will be performed after the above two steps in order to highlight the potential discrepancies and prepare the recommended corrective measures. Recommendation can be given to design and coding rules.

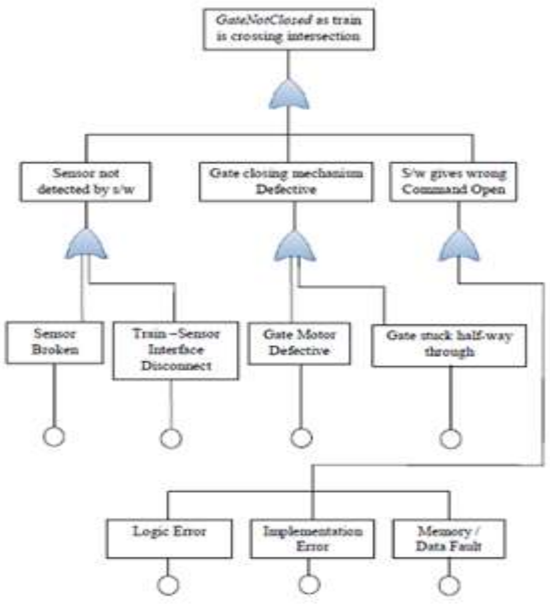
SFMEA Analysis of RCCS

The SFMEA, a sample of which is shown in the Table 2 below presents some software failure modes defined for RCCS. The origin and effects of each failure mode are analyzed identifying the top level events for further refinement, when the consequence of this failure could be catastrophic for this system. Three top events were singled out for further analysis of failure mode Gate not closed as train is passing through railroad intersection.

Failure Mode	**Possible Causes**	**Effect**	**Sever-ity of risk**	**Prevention And Compensation**
Gate not closed as train is passing through	a) sensor not detected by s/w b) gate motor mechanism is defective c) s/w gives wrong command d) s/w gives right command at wrong time	Train collision with passing road traffic leading to accidents	Critical	Software first checks the working status of gates each time the train is about to cross the gates
Track change lever is not acti-vated to change train route	a) sensor is not detected by s/w b) track lever motor mechanism is defective c) s/w gives wrong command to lever d) s/w gives right command at wrong time e) s/w fails to give a command to acti-vate lever	Train fails to change its path from the outer track circuit to the inner track circuit leading to accident	Critical	Software first checks the working status of the track lever each time the train is about to enter the inner track loop
Control program software is corru-pted	a) logic fault b) interface fault c) data fault d) calculation fault e) memory fault	Unpredictable sequence of opera-tion leading to accident	Critical or Catast-rophic	Algorithm logic is verified for accuracy. Data Structures and Memory overflow is checked.

SFTA Analysis of RCCS

The fault tree is a graphical representation of the conditions or other factors causing or contributing to the occurrence of the so-called top event, which normally is identified as an undesirable event. A systematic construction of the fault tree consists in defining the immediate cause of the top event. These immediate cause events are the immediate cause or immediate mechanism for the top event to occur. From here, the immediate events should be considered as sub-top events and the same process should be applied to them. All applicable fault types should be considered for applicability as the cause of a higher level fault. This process proceeds down the tree until the limit of resolution of tree is reached, thereby reaching the basic events, which are the terminal nodes of the tree.


GateNotClosed as train is crossing intersection
Sensor not detected by s/w
Gate closing mechanism Defective
S/w gives wrong Command Open
Sensor Broken
Train –Sensor Interface Disconnect
Gate Motor Defective
Gate stuck half-way through
Logic Error
Implementation Error
Memory / Data Fault

8. CONCLUSION

Verifying safety-critical software is an important activity during safety-related software development. Unfortunately, there are no methodology frameworks for carrying out this activity; therefore, this research developed a methodology framework for statically and dynamically verifying safety-critical software systems.

The methodology follows a life-cycle approach to verification by supplying methods and guidelines for preliminary hazard analysis, high-level-design hazard analysis, detailed-design hazard analysis, and code-level hazard analysis. Furthermore, the methodology contains several testing and coverage techniques along with guidelines for dynamic verification, which is an area that research largely ignores in spite of its importance.

References

[1] MIl-STD-1574A (USAF)**,** “System Safety Program for Space and Missile Systems,” Dept of Defense, US Govt. Printing Office, 1979

[2] P. V. Bhansali, “Software Safety: Current Status and Future Directions” *ACM SIGSOFT Software Engineering Notes*, Volume 30 Number 1, page 1, January 2005

[3] John C. Knight, “Safety Critical Systems: Challenges and Directions*”, Proceedings of the 24*th *International Conference on Software Engineering (ICSE)*, Orlando, Florida, 2002

[4] MIL-STD-882C, System Safety Program Requirements 1993, http://eic.ipo.noaa.gov/IPOarchive/MAN /doc124.pdf

[5] N.G. Leveson, Safeware: *System Safety and Computers*, Addison-Wesley, Reading, MA, USA, 1995.

[6] W.R. Dunn, Practical Design of Safety-Critical *Computer Systems*, Reliability Press, 2002.

[7] M.S. Jaffe and N.G.Leveson, “Completeness, robustness, and safety in real-time software requirements specification”, *Proc. of the 11th International Conference on Software engineering (ICSE)*, Pittsburgh, USA, pp. 302-311, 1989

[8] Raghu Singh. “A Systematic Approach to Software Safety”. *Proceedings of Sixth Asia Pacific Software Engineering Conference* (APSEC), Takamatsu, Japan ,1999.

[9] T. Shimeall and N. Leveson, *An Empirical Comparison of Software Fault Tolerance and Fault Elimination*, IEEE Transactions On Software Engineering, vol. SE-17, no. 2, pp. 173-183, 1991.

[10] P. Rodríguez Dapena. ‘How are static fault removal techniques verifying software safety and reliability?’ Joint ESA-NASA Space-FlightSafety Conference. ESA. 06-Nov-2001

[11] *Software Safety*. NASA-STD-8719.13A NASA Technical Standard. September 15, 1997 Replaces NSS 1740.13 dated February 1996. http://swg.jpl.nasa.gov/resources/index.shtml

[12] L. M. Ippolito, D. R. *Wallace A Study on Hazard Analysis in High Integrity Software Standards and Guidelines*. National Institute of Standards and Technology. NIST IR 5589. January 1995 .